Observer Pattern

**Observer Pattern’in Temel Mantığı**

Observer Pattern, yazılım dünyasında "publish-subscribe" (yayınla-abone ol) olarak da bilinir. Bu desenin amacı, bir nesnenin (Subject veya Publisher) durumundaki değişikliklerin, o nesneyi izleyen diğer nesnelere (Observer veya Subscriber) otomatik olarak bildirilmesidir.

#### Roller ve Bileşenler

**Subject (Publisher):** Gözlemlenen nesnedir. Bu nesne, diğer nesnelerin (observer) abone olduğu bir yapı sağlar ve kendi içsel durumunda bir değişiklik olduğunda tüm gözlemcilere haber verir.

**Observer (Subscriber):** Subject'i izleyen nesnedir. Subject'e abone olduktan sonra, subject'in durumundaki değişikliklerden haberdar olur ve ona göre kendi içsel durumunu günceller.

**Concrete Subject (Somut Subject):** Subject soyut bir sınıf ya da arayüz olarak tanımlanabilir ve bunun somut hali, durum değişikliği gerçekleştiren nesnedir. Bu nesne, gözlemcileri bir liste halinde tutar ve değişiklik olduğunda hepsini bilgilendirir.

**Concrete Observer (Somut Observer):** Observer arayüzünü ya da soyut sınıfını implemente eden sınıftır. Subject'ten gelen bildirimlere göre tepki verir ve kendi iç durumunu günceller.

### Observer Pattern'in Avantajları

**Gevşek Bağlantı:** Subject ve Observer nesneleri arasında düşük bağlılık (loose coupling) sağlar. Subject gözlemciler hakkında pek bir bilgiye sahip olmadan, yalnızca onları bilgilendirme görevini yerine getirir.

**Esneklik:** Yeni observer nesneleri eklemek veya mevcut olanları kaldırmak oldukça kolaydır. Subject ve Observer’lar arasında doğrudan bir bağ olmadığından, bu yapıyı değiştirmek minimal etki yaratır.

**Dinamik Güncellemeler:** Observer'lar subject’in durum değişikliklerine otomatik olarak yanıt verebilir. Herhangi bir bileşen değişikliğinde tüm gözlemcilerin güncel kalması sağlanır.

**Reaktif Programlama:** Bu tasarım deseni, reaktif programlamanın bir temelidir. Nesneler sürekli olarak subject’in durumuna tepki verirler.

### Observer Pattern Adımları

### *Subject Tarafı (Gözlemlenen)*

Subject, gözlemcilerin abone olduğu ve durum değişikliği olduğunda onları bilgilendiren kısımdır. Bu yapı genellikle aşağıdaki adımları içerir:

* Bir observer listesi tutar.
* Gözlemcilerin bu listeye abone olabilmesini sağlar (register).
* Gözlemcileri bu listeden çıkarabilme yetisi sunar (unregister).
* Subject’in durumu değiştiğinde, listedeki tüm gözlemcilere durumu bildirir (notify).

#### 2. Observer Tarafı (Gözlemci)

Observer, subject'in durumunu izleyen kısımdır. Observer, subject’e abone olur ve subject’te bir değişiklik olduğunda tepki gösterir. Observer'lar, subject'ten gelen güncellemeleri alır ve ona göre kendi işlemlerini yaparlar.

### React Native’de Observer Pattern Kullanımı

React Native'de Observer Pattern’i kullanmanın çeşitli yolları vardır. State yönetimi kütüphaneleri, reaktif veri akışı ve bileşenlerin durum bazlı yeniden render edilmesi Observer Pattern'in birer örneğidir.

#### MobX İle Detaylı Observer Pattern Uygulaması

MobX, React Native’de Observer Pattern’in uygulandığı popüler bir state yönetim kütüphanesidir. MobX'in temel mantığı, bir bileşeni observable (gözlemlenebilir) hale getirip, değişiklik olduğunda bu değişikliğin gözlemlenip otomatik olarak bileşenin yeniden render edilmesini sağlamaktır.

##### Adım Adım MobX ile Kullanım

1. **Observable State (Gözlemlenebilir Durum):** mobx kütüphanesi ile bir state’i gözlemlenebilir hale getiririz. Aşağıdaki örnekte, bir sayacın (counter) gözlemlenebilir bir state olarak tanımlanması gösteriliyor.

A computer screen shot of a code

Description automatically generated

**Observer Bileşeni (Gözlemci Bileşen):**

Observer bileşeni, observer fonksiyonu ile oluşturulur. Bu fonksiyon sayesinde, bileşen observable state’e abone olur ve state değiştiğinde yeniden render edilir.

A computer screen shot of code

Description automatically generated

1. **Gözlemleme İşlemi:**

Bu kod çalıştığında, counterStore içindeki count değeri her değiştiğinde CounterComponent bileşeni yeniden render edilir. Bu sayede, bileşen otomatik olarak count değerini izler ve her güncellemede arayüzde değişiklik gösterilir.

Factory Pattern

**Factory Pattern**, yazılım geliştirmede kullanılan **creational (yaratıcı) tasarım desenlerinden** biridir. Amacı, nesnelerin oluşturulma sürecini yönetmektir. Bir nesneyi doğrudan sınıfını kullanarak oluşturmak yerine, bu desende bir **fabrika (factory)**, belirli koşullara göre nesneleri oluşturur ve nesnenin oluşturulma sürecini soyutlar. Bu, nesnelerin nasıl ve ne zaman oluşturulacağına dair esnekliği artırır ve kodun bakımını kolaylaştırır.

### Factory Pattern'in Amacı ve Avantajları

1. **Soyutlama**: Nesne oluşturma sürecini soyutlar, böylece oluşturulan nesnenin tipini ve nasıl oluşturulduğunu bilmek zorunda kalmazsınız.
2. **Modülerlik**: Eğer oluşturulacak nesneye dair bir değişiklik yapmanız gerekirse, sadece fabrika sınıfında değişiklik yaparak nesne oluşturma işlemini kolayca yönetebilirsiniz.
3. **Esneklik**: Kodunuzu daha esnek ve genişletilebilir hale getirir. Eğer yeni bir tür nesne eklemek isterseniz, fabrika sınıfına yeni bir seçenek ekleyebilirsiniz.
4. **Kod Tekrarını Azaltır**: Aynı türde nesneler oluşturulurken tekrar eden kodları tek bir noktada toplar, bu da kod tekrarını azaltır.

### Factory Pattern’in Yapısı

Factory Pattern’de genellikle iki temel bileşen bulunur:

1. **Factory (Fabrika)**: Nesneleri oluşturmak için kullanılan sınıftır. Bu sınıf, parametrelere veya duruma bağlı olarak farklı tipte nesneler oluşturur.
2. **Product (Ürün)**: Bu, fabrika tarafından oluşturulan nesnedir. Aynı arabirimi veya abstract sınıfı paylaşan nesneler olabilir.

### Factory Pattern Kullanım Adımları

1. Bir **interface** ya da **abstract class** tanımlayarak, oluşturulacak nesnelerin ortak bir arayüzünü belirlersiniz.
2. Farklı ürün türlerini bu interface ya da abstract class’tan türetirsiniz.
3. Bir **Factory class** tanımlarsınız. Bu sınıf, hangi nesnenin oluşturulacağını belirleyip o nesneyi döndürür.

### React Native’de Factory Pattern

React Native'de Factory Pattern, özellikle **dinamik olarak bileşenler oluşturmanız gerektiğinde** ya da **farklı koşullara göre bileşenlerin yapısını değiştirmeniz gerektiğinde** kullanışlı olabilir. Örneğin, UI bileşenlerini, API'den gelen verilere göre farklı türlerde oluşturmanız gereken bir uygulama senaryosu olabilir.

#### 1. Örnek Senaryo: Buton Üretim Fabrikası

Diyelim ki uygulamanızda birçok farklı tipte buton bulunuyor ve her buton farklı özelliklere sahip. Factory Pattern ile bu butonları oluşturan bir fabrika tanımlayabilirsiniz. Her buton için ayrı bir bileşen oluşturmaktansa, dinamik olarak buton tipine göre bir bileşen oluşturabilirsiniz.

##### Adım 1: Ortak bir Interface veya Sınıf Tanımlama

Tüm buton bileşenleri için bir yapı belirleyelim. Bu yapıya ButtonComponent diyelim.

A computer screen shot of a program code

Description automatically generated

##### Adım 2: Farklı Buton Türleri İçin Bileşenler

Şimdi farklı türlerde buton bileşenlerini tanımlayalım. Örneğin, **Primary**, **Secondary**, ve **Danger** butonları olsun.

A computer screen shot of text

Description automatically generated

##### Adım 3: Butonları Üretecek Factory Sınıfı

Bir buton fabrika sınıfı oluşturalım ve parametrelere göre doğru buton bileşenini oluşturalım. Bu fabrika, hangi butonun oluşturulacağına karar verir.

A computer screen shot of a program code

Description automatically generated

##### Adım 4: Factory Pattern'in Kullanımı

Fabrika sınıfını uygulamada kullanarak dinamik olarak butonlar oluşturabiliriz. Bir buton oluşturmak için ButtonFactorykullanabiliriz.

A computer screen shot of a program code

Description automatically generated

Bu örnekte, **ButtonFactory**'yi kullanarak buton türüne göre doğru bileşeni oluşturduk. Böylece uygulamanızda farklı buton türlerini dinamik olarak üretebilirsiniz.

### Factory Pattern'in Faydaları

1. **Dinamik Bileşen Üretimi:** Factory Pattern, aynı bileşen ailesine ait ancak küçük farklara sahip bileşenleri (örneğin, Primary ve Secondary butonlar) dinamik olarak üretmenizi sağlar.
2. **Kod Tekrarını Azaltır:** Aynı türde birden fazla bileşen oluştururken her bileşeni ayrı ayrı kodlamak yerine, bir fabrika aracılığıyla bu bileşenlerin oluşturulması kod tekrarını azaltır.
3. **Bakım Kolaylığı:** Yeni bir buton türü veya bileşen eklemek istediğinizde sadece fabrika fonksiyonunu güncelleyerek tüm kod yapısını bozmadan bu bileşeni ekleyebilirsiniz.
4. **Test Edilebilirlik:** Nesneleri fabrika sınıfı üzerinden oluşturmak, testlerde hangi nesnelerin ne zaman üretildiğini kontrol etmenizi kolaylaştırır. Factory Pattern, test edilebilirliği artırır.

### Factory Pattern Kullanım Alanları

1. **UI Bileşenleri Üretimi:** Farklı stillerdeki UI bileşenlerini (butonlar, giriş alanları, kartlar, vb.) dinamik olarak üretmek istediğinizde.
2. **API Verilerine Göre Bileşen Oluşturma:** Kullanıcıdan ya da API'den gelen verilere göre bileşen yapısını dinamik olarak değiştirmek istiyorsanız.
3. **Çoklu Platform Desteği:** React Native’de iOS ve Android’e göre değişen bileşenler üretmek istediğinizde. Örneğin, platforma özgü butonlar veya UI bileşenlerini oluşturmak için Factory Pattern kullanılabilir.

### Özetle;

**Factory Pattern**, React Native’de, özellikle UI bileşenleri oluşturma veya API’den gelen verilere göre farklı tipte bileşenler üretme gibi durumlarda oldukça faydalıdır. Bu desen, kodu daha modüler, esnek ve genişletilebilir hale getirir, ayrıca kod tekrarını önler. Factory Pattern ile kullanıcı arayüzü öğelerini dinamik olarak üretmek ve yönetmek çok daha düzenli ve sürdürülebilir bir hale gelir.

Birkaç pattern örneği daha:

### Render Props Pattern

Render Props deseni, bir bileşenin render edilme sürecini kontrol etmek için fonksiyon olarak bir prop kullanmayı ifade eder. Yani bir bileşen, kendi içindeki veriyi ya da durumu dışarıdan gelen bir fonksiyon aracılığıyla işleyip görüntüleyebilir. Bu desenin temel amacı, aynı mantığı birden fazla bileşende tekrar kullanabilmektir. Örneğin, bir bileşen fare pozisyonunu izliyor olabilir ve farklı UI ihtiyaçlarına göre bu pozisyonu görüntülemek isteyebilirsiniz. Render Props, bu tür durumlarda kullanılabilecek esnek bir çözümdür.

**Nasıl çalışır?**

* Bir bileşene prop olarak bir fonksiyon (genelde "render" olarak adlandırılır) geçirilir.
* Bu fonksiyon, bileşenin içindeki durumu (örneğin state veya props) kullanarak UI’ın nasıl görüneceğini belirler.
* Böylece aynı bileşeni, farklı fonksiyonlar geçirerek farklı şekillerde render edebilirsiniz.

**Avantajlar:**

* Bileşenler arası mantık paylaşımı sağlar.
* UI’ın esnek bir şekilde kontrol edilmesini mümkün kılar.
* Yeniden kullanılabilir ve daha modüler yapılar oluşturmayı kolaylaştırır.

### Decorator Pattern

Decorator deseni, mevcut bir bileşene ek işlevsellik kazandırmak için kullanılan bir yöntemdir. Bu desenin temel fikri, bir bileşeni sarmalayarak ona yeni özellikler eklemektir. Örneğin, bir bileşeni bir başka bileşenle sararak ona animasyonlar, ek stil ya da olay yönetimi (örneğin tıklama olayları) ekleyebilirsiniz.

**Nasıl çalışır?**

* Bir bileşen, başka bir bileşen tarafından sarılır ve ek işlevler bu sarma işlemi sırasında uygulanır.
* Sarılan bileşen orijinal işlevini korur, ancak dekorator tarafından eklenen yeni işlevlerle genişletilir.
* Bu yaklaşım, bileşenlerin dinamik olarak özellik kazanmasını sağlar.

**Avantajlar:**

* Kodunuzu daha modüler hale getirir, çünkü aynı bileşene farklı işlevler ekleyebilirsiniz.
* Aynı bileşeni birçok farklı durumda tekrar tekrar kullanmanıza olanak tanır.
* Stil, animasyon veya olay yönetimi gibi işlevsellikleri kolayca eklemeyi sağlar.

### Command Pattern

Command deseni, bir işlemi bir komut nesnesine dönüştürerek, bu komutları gerektiği anda çalıştırmayı sağlar. Bu desen, genellikle kullanıcı etkileşimleri veya karmaşık işlemleri soyutlamak için kullanılır. Örneğin, bir butona tıklanması sonucu belirli bir işlemi gerçekleştirmek istediğinizde, bu işlemi bir komut nesnesine dönüştürüp daha sonra tetikleyebilirsiniz.

**Nasıl çalışır?**

* Bir işlem, komut olarak soyutlanır ve bu komut nesnesi belirli bir işlevselliği içerir.
* Bu komut nesneleri tetiklenene kadar bağımsız bir şekilde durur.
* İhtiyaç olduğunda, bu komutlar çalıştırılarak işlemler gerçekleştirilir.
* Bu yaklaşım, karmaşık işlemleri yönetmeyi kolaylaştırır.

**Avantajlar:**

* İşlemleri daha esnek ve dinamik bir şekilde yönetmenizi sağlar.
* Karmaşık kullanıcı etkileşimlerini daha düzenli bir şekilde ele alabilirsiniz.
* İşlemlerinizi daha modüler hale getirir, çünkü her işlem bir komut olarak soyutlanır.

### Strategy Pattern

Strategy deseni, belirli bir işlemi gerçekleştirmek için birden fazla strateji tanımlama ve bu stratejiler arasında geçiş yapabilme yeteneği sunar. Bu desen, farklı algoritmaların veya yöntemlerin aynı işlem için kullanılabileceği durumlarda çok kullanışlıdır. Örneğin, sıralama işlemleri gibi işlemler farklı stratejilerle (artan ya da azalan) yapılabilir ve bu stratejiler arasında runtime’da geçiş yapılabilir.

**Nasıl çalışır?**

* Bir işlem için çeşitli stratejiler tanımlanır. Örneğin, bir sıralama işlemi için "artan sıralama" ve "azalan sıralama" stratejileri olabilir.
* Bu stratejiler, runtime sırasında ihtiyaçlara göre seçilip uygulanabilir.
* Kullanıcı duruma göre farklı stratejiler arasında geçiş yapabilir.

**Avantajlar:**

* Farklı algoritmaların esnek bir şekilde kullanılmasını sağlar.
* Stratejiler arasında dinamik geçiş yapmayı kolaylaştırır.
* Aynı işlemi farklı şekillerde gerçekleştirebilme olanağı sunar.